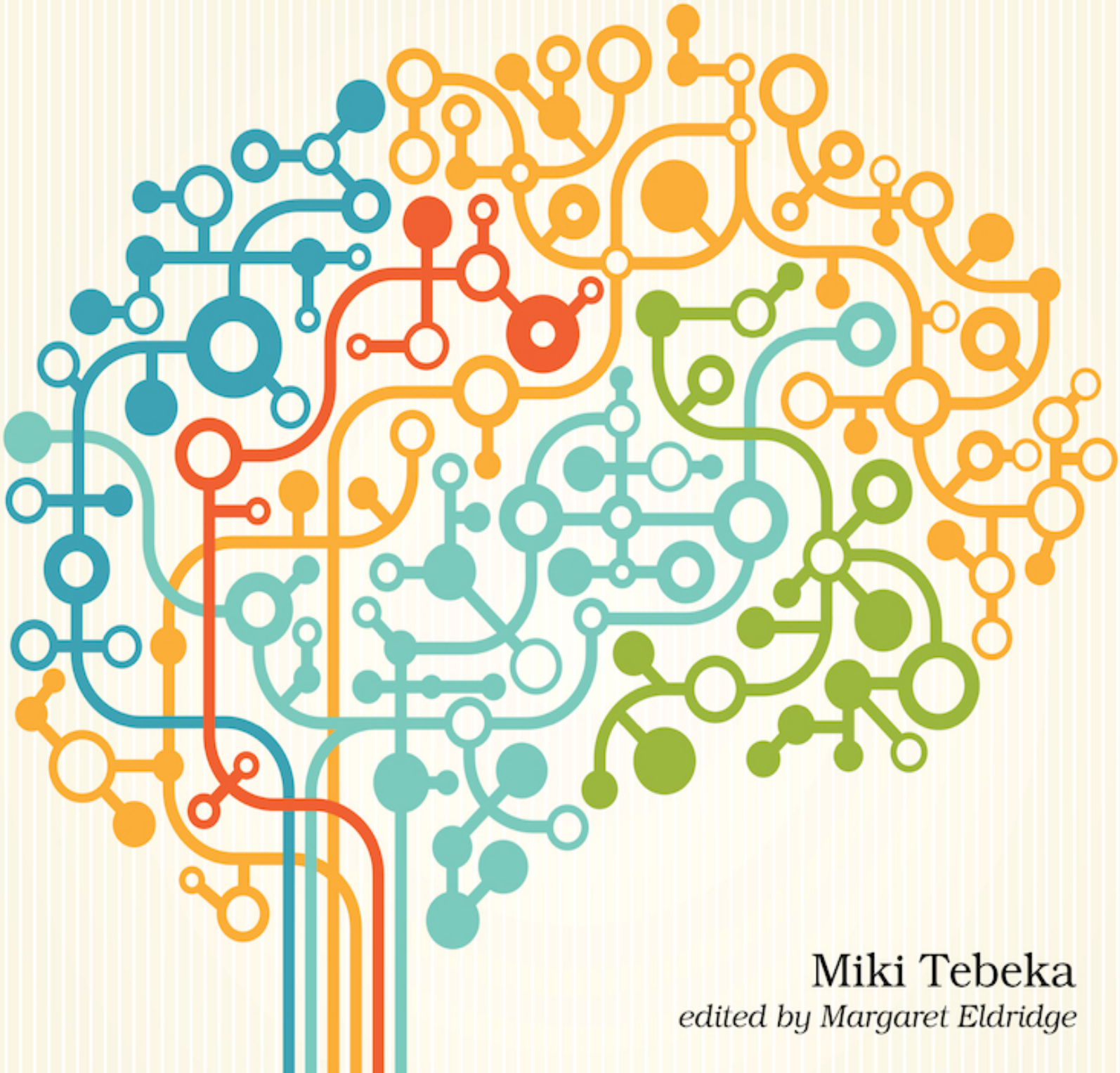
The Pragmatic Programmers

Go Brain Teasers

Exercise Your Mind



Miki Tebeka

edited by Margaret Eldridge

Go Brain Teasers

Exercise Your Mind

by Miki Tebeka

Version: P1.0 (September 2021)

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://pragprog.com.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it's still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We're here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's polices. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/#about-ebooks. To learn more about this book and access the free resources, go to https://pragprog.com/book/d-gobrain, the book's homepage.

Thanks for your continued support,

Andy Hunt
The Pragmatic Programmers

The team that produced this book includes: Dave Rankin (CEO), Janet Furlow (COO), Tammy Coron (Managing Editor), Margaret Eldridge (Development Editor), Jennifer Whipple (Copy Editor), Potomac Indexing, LLC (Indexing), Gilson Graphics (Layout), Andy Hunt and Dave Thomas (Founders)

For customer support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

To Sharon, who suffered me in quarantine, and the twenty years before that.

Table of Contents

Early Praise for *Go Brain Teasers*

In *Go Brain Teasers*, Miki has prepared a set of tests that seek not to simply baffle but to enlighten. *Go Brain Teasers* provides the reader with the opportunity to learn the *why* behind that *what!?!*

→ Dave Cheney
Staff Software Engineer, GitHub

Go Brain Teasers should be renamed *Go Mind Blowers*.

→ David Bordeynik
Software Architect, NVIDIA

Go Brain Teasers is the completion for the "Tour of Go" that you were looking for. It suits anyone who is writing Go and wants to grasp the language in a fun and productive way.

→ Yoni Davidson
Software Engineer and Data Architect, Bond

I see after having started reading that the exercises are very well-explained. Thanks! Been enjoying your book, by the way.

→ Jared Davis
Owner / Software Engineer, 1ijk Development, LLC

Acknowledgments

I’m grateful for every contribution, from finding bugs to fixing grammar to letting me work in peace.

Here is a list of people who helped; my apologies to anyone I forgot:

- Adi Tebeka for her proofreading and comments.
- Dan Allen for his help in the asciidoctor forums. I used the wonderful asciidoctor to write the initial version of this book.
- Dave Cheney for the foreword.
- David Bordeynik for his comments.
- Egon Elbre who drew all these wonderful gophers and placed them in CC0 license.
- Elad Eyal for proofreading.
- Eliran Bivas for proofreading and comments.
- Jared David for his quote.
- Ran Tavory for his comments.
- Yoni Davidson for his comments.

I’m also grateful to Brad Fitzpartick and others who post Go “pop quizzes” and gave me many ideas for these brain teasers.

Preface

The Go programming language is a simple one, but like all other languages, it has its quirks. This book uses these quirks as a teaching opportunity. By understanding the gaps in your knowledge, you'll become better at what you do.

There's a lot of research showing that people who make mistakes during the learning process learn better than people who don't. If you use this approach at work when fixing bugs, you'll find you enjoy bug hunting more and become a better developer after each bug you fix.

These teasers will help you avoid mistakes. Some of the teasers are from my own experience shipping bugs to production, and some are from others doing the same.

Teasers are fun! We geeks love puzzles and solving them. You can also use these teasers to impress your coworkers, have knowledge competitions, and become better together.

Many of these brain teasers are from quizzes I gave at conferences and meetups. I've found that people highly enjoy them, and they tend to liven the room.

At the beginning of each chapter, I'll show you a short Go program and ask you to guess the output. The possible answers can be the following:

- Won't compile

- Panic
- Deadlock
- Some output (e.g., `[1 2 3]`)

Go Version

I've used Go version 1.14.1 to run the code; the output *might* change in future versions.

Before moving on to the answer and the explanation, go ahead and guess the output. After guessing the output, I encourage you to run the code and see the output yourself; only then proceed to read the solution and the explanation. I've been teaching programming for many years and found this course of action to be highly effective.

About the Author

Miki Tebeka has a B.Sc. in computer science from Ben Gurion University. He also studied there toward an M.Sc. in computational linguistics.

Miki has a passion for teaching and mentoring. He teaches many workshops on various technical subjects all over the world and has mentored many young developers on their way to success. Miki is involved in open source and has several projects of his own and contributed to several more, including the Python project. He has been using Python for more than twenty-three years.

Miki wrote *Pandas Brain Teasers*, *Python Brain Teasers*, and *Forging Python* and is a LinkedIn Learning author and an organizer of Go Israel Meetup, GopherCon Israel, and PyData Israel Conference.

About the Code

You can find the brain teasers code at https://pragprog.com/titles/d-gobrain/go-brain-teasers/.

I've tried to keep the code as short as possible and remove anything that is not related to the teaser. This is *not* how you'll normally write code.

About You

I assume you know Go at some level and have experience programming with it. This book is not for learning how to program in Go. If you don't know Go, I'm afraid these brain teasers are not for you.

One More Thing

As you work through the puzzles in this book, it might help to picture yourself as Nancy Drew, Sherlock Holmes, or any other of your favorite detectives trying to solve a murder mystery in which *you* are the murderer. Think of it like this:

> Debugging is like being a detective in a crime movie where you're also the murderer.
>
> — Filipe Fortes

With this mindset, I have found that things are easier to understand, and the work is more enjoyable. So, with that in mind, have fun guessing the brain teasers in this book—perhaps you might even learn a new trick or two.

If you'd like to learn more, please send an email to mailto:info@353solutions.com, and we'll tailor a hands-on workshop to meet your needs. There's also a comprehensive offering of hands-on workshops at https://www.353solutions.com.

Stay curious, and keep hacking!

Miki Tebeka, March 2020

Foreword by David Cheney

As a fan of trivia, in the salad days of my education as a software engineer, one of my favorite books was Josh Bloch and Neal Gafter's *Java Puzzlers*. I liked that the authors didn't simply set out to stump readers with obscure language factoids. Instead, each question was treated as an opportunity to educate the reader on the history and deeper meaning of a less-traveled aspect of the language.

When I discovered, far too many years into my Go experience than I care to mention, that `copy` returned the number of elements copied, my first thought was "Wow, how had I missed that?" My second thought was "I wonder how many people I can trick with this." Thus was born my #golang pop quiz series of tweets.

With tweet sizes being what they are, the requirement to fit an entire Go program into a single tweet proved a challenge. Divining the correct answer to a #golang pop quiz and the opportunity to follow Bloch and Gafter's example to educate, rather than frustrate, was left as an exercise to the reader.

In *Go Brain Teasers*, Miki has prepared a set of tests that seek not to simply baffle but to enlighten. *Go Brain Teasers* provides the reader with the opportunity to learn the *why* behind that *what!?!*

David Cheney
Sydney, April 2020

Part 1
Go Brain Teasers

Puzzle 1 A Number π

pi.go

```
package main

import (
        "fmt"
)

func main() {
        var π = 22 / 7.0
        fmt.Println(π)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `3.1415929203539825`

There are two surprising things here: one is that `π` is a valid identifier, and the second is that `355 / 113.0` actually compiles.

Let's start with `π`. Go language specification on identifiers says

> Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

Letters can be Unicode letters, including `π`. This can be fun to write, but in practice it'll make your coworkers' lives harder. I can easily type `π` with the

Vim editor I'm using; however, most editors and IDEs will require more effort.

The only place I've found where Unicode identifiers are helpful is when translating mathematical formulas to code. Apart from that, stick to plain old ASCII.

Now to `355 / 113.0`. The Go type system will not allow dividing (or any other mathematical operation) between an integer (`355`) and a float (`113.0`). But what you have on the right side of the `=` are constants, not variables. The type of constant is defined when it is being used; in this example, the compiler will convert `355` to a float to complete the operation.

If you first assign `355` and `113.0` to variables and try the same code, it will fail. The following won't compile.

pi_var.go

```
package main

import (
        "fmt"
)

func main() {
        a, b := 22, 7.0
        var n = a / b
        fmt.Println(n)
}
```

Further Reading

Constants Specification
http://golang.org/ref/spec#Constants

Constants
http://blog.golang.org/constants

"Introduction to Numeric Constants in Go" by Ardan Labs
http://ardanlabs.com/blog/2014/04/introduction-to-numeric-constants-in-go.html

Approximations of π on Wikipedia
http://en.wikipedia.org/wiki/Approximations_of_%CF%80

Go Language Specification on Identifiers
http://golang.org/ref/spec#Identifiers

Puzzle 2 Empty-Handed

empty_map.go

```
package main

import (
        "fmt"
)

func main() {
        var m map[string]int
        fmt.Println(m["errors"])
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `0`

The zero value for an uninitialized map is `nil`. Some operations on Go's map type are `nil` safe, meaning they will work with a `nil` map without panicking.

Both `len` and accessing a value (e.g., `m["errors"]`) will work on a `nil` map. `len` returns `0`, and accessing a value by a key will do the following:

- If the key is in the map, return the value associated with it.
- If the key is not in the map, return the zero value for the value type.

In this example, the key `"errors"` is not found in the map and you'll get back the zero value for `int`, which is `0`.

This behavior is handy. If you want to count items (say, word frequency in a document), you can do `m[word]++` without worrying if `word` is in the map `m` or not.

This leads to the question, How can you know if a value you got from a map is because it's there or because it's the zero value for the value type? The answer is to use the `comma, ok` paradigm.

When you type `val, ok := m[key]`, `ok` will be `true` if we got the value from the map and `false` if `key` is not in the map.

There are other `nil` safe types in Go. For example, you can ask the length of a `nil` slice, receive a value from an empty channel (though you'll be blocked forever), and more.

Further Reading

Go Maps in Action
http://blog.golang.org/maps

Maps in Effective Go
http://golang.org/doc/effective_go.html#maps

Zero Value in the Go Specification
http://golang.org/ref/spec#The_zero_value

Puzzle 3 When in Kraków

city.go

```
package main

import (
        "fmt"
)

func main() {
        city := "Kraków"
        fmt.Println(len(city))
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `7`

If you count the number of characters in `Kraków`, it'll come out to six. So why 7? The reason is ... history.

In the beginning, computers were developed in English-speaking countries: the UK and the US. When early developers wanted to encode text in computers that only understood bits, they came out with the following scheme. Use a `byte` (8 bits) to represent a character. For example `a` is 97 (`01100001`), `b` is 98, and so on. One byte is enough for the English alphabet, containing twenty-six lowercase letters, twenty-six uppercase letters, and ten digits. There is even some space left for other special characters (e.g., 9 for tab). This is known as *ASCII encoding*.

After a while, other countries started to use computers, and they wanted to write using their native languages. ASCII wasn't good enough; a single byte isn't big enough to hold all the numbers needed to represent letters in different languages. This led to several different encoding schemes; the most common one is UTF-8. Rob Pike, one of the designers of Go, is also one of the designers of UTF-8.

Go strings are UTF-8 encoded. This means that a character (called *rune*) can be from 1 to 4 bytes long. When you ask the length of a string in Go, you'll get the size in bytes. In this example, the rune `ó` is taking 2 bytes; hence, the total length of the string is 7.

If you want to know the number of runes in a string, you'll need to use the `unicode/utf8` package.

cityu.go

```
package main

import (
        "fmt"
        "unicode/utf8"
)

func main() {
        city := "Kraków"
        fmt.Println(utf8.RuneCountInString(city))
}
```

Further Reading

Strings, Bytes, Runes, and Characters in Go
http://blog.golang.org/strings

Unicode and You
http://betterexplained.com/articles/unicode/

Unicode on Wikipedia

http://en.wikipedia.org/wiki/Unicode

ASCII on Wikipedia
http://en.wikipedia.org/wiki/ASCII

UTF-8 on Wikipedia
http://en.wikipedia.org/wiki/UTF-8

Puzzle 4 I've Got Nothing

nil.go

```
package main

import (
        "fmt"
)

func main() {
        n := nil
        fmt.Println(n)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will not compile.

`nil` is not a type but a reserved word. A variable initialized to `nil` must have a type. If, for example, you change the assignment to `var n *int = nil`, the code will compile since now `n` has a type.

Here are some places where `nil` is used:

- The zero value for `map`, slice, and `chan` is `nil`.
- You can't compare slices or maps using `==`; you can only compare them to `nil`.

- Sending to or receiving from a `nil` channel will block forever. You can use this to avoid a busy wait.

Further Reading

"Understanding `nil`" from Practical Go by Dave Cheney
http://dave.cheney.net/practical-go/presentations/gophercon-israel.html#nil

Why Are There `nil` Channels in Go?
http://medium.com/justforfunc/why-are-there-nil-channels-in-go-9877cc0b2308

Channel Axioms by Dave Cheney
http://dave.cheney.net/2014/03/19/channel-axioms

Puzzle 5 A Raw Diet

raw.go

```
package main

import (
        "fmt"
)

func main() {
        s := `a\tb`
        fmt.Println(s)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `a\tb`

The Go spec[1] says

> There are two forms: raw string literals and interpreted string literals.

Raw strings are enclosed in backticks interpreted strings are enclosed in quotes. In raw strings, `\` has no special meaning, so the `'\t'` is two characters, backslash and the letter `t`. If it was an interpreted string, `\t` would be interpreted as the tab character.

Apart from the usual `\t` (tab), `\n` (newline), and friends, you can use Unicode code points in interpreted strings. `fmt.Println("\u2122")` will print ™.

One of the most common uses for raw strings is to create multiline strings.

```
// An HTTP request
request := `GET / HTTP/1.1
Host: www.353solutions.com
Connection: Close

`
```

Further Reading

Go Spec on Literals
http://golang.org/ref/spec#String_literals

Strings, Bytes, Runes, and Characters in Go
http://blog.golang.org/strings

String Spec
http://golang.org/ref/spec#String_literals

ASCII Control Characters on Wikipedia
http://en.wikipedia.org/wiki/Control_character#In_ASCII

Puzzle 6 Are We There Yet?

time.go

```
1  package main
2
3  import (
4          "fmt"
5          "time"
6  )
7
8  func main() {
9          timeout := 3
10         fmt.Printf("before ")
11         time.Sleep(timeout * time.Millisecond)
12         fmt.Println("after")
13 }
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will not compile.

When you write `timeout := 3`, the Go compiler will do a type inference. In this case it will infer that `timeout` is an `int`. Then in line 11, you multiply `timeout` (`int` type) with `time.Millisecond` (which is of `time.Duration` type), which is not allowed.

You have several options to fix this.

Change the type of `timeout` to `time.Duration` (line 9):

```
var timeout time.Duration = 3
```

Make `timeout` a `const`, and then its type will be resolved in the context of usage (line 9):

```
const timeout = 3
```

Use a type conversion to convert `timeout` to `time.Duration` (line 11):

```
time.Sleep(time.Duration(timeout) * time.Millisecond)
```

Further Reading

Type Inference on Wikipedia
http://en.wikipedia.org/wiki/Type_inference

Type Inference in the Go Tour
http://tour.golang.org/basics/14

go/types: The Go Type Checker in golang/example
https://github.com/golang/example/tree/master/gotypes

time.Duration
http://golang.org/pkg/time/#Duration

Type Conversions in the Go Specification
http://golang.org/ref/spec#Conversions

Puzzle 7 Can Numbers Lie?

float.go

```
package main

import (
        "fmt"
)

func main() {
        n := 1.1
        fmt.Println(n * n)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `1.2100000000000002`

You might have expected `1.21`, which is the right mathematical answer.

Some new developers, when seeing this or similar output, come to the message boards and say, “We found a bug in Go!” The usual answer is, “Read the fine manual” (RTFM).

> Floating point is sort of like quantum physics: the closer you look, the messier it gets.
>
> — Grant Edwards

The basic idea behind this issue is that in floating points, we sacrifice accuracy for speed (i.e., cheat). Don't be shocked; it's a trade-off we do a lot in computer science.

The result you see conforms with the floating-point specification. If you run the same code in Python, Java, C … you will see the same output.

See the links in the following section if you're interested in understanding more about how floating points work. The main point you need to remember is that they are not accurate, and accuracy worsens as the number gets bigger.

One implication is that when testing with floating points, you need to check for "roughly equal" and decide what is an acceptable threshold. Testing libraries such as `stretchr/testify` have ready-made functions (such as `InDelta`) to check that two floating numbers are approximately equal.

Floating points have several other oddities. For example, there's a special `NaN` value (short for *not a number*). `NaN` does not equal any number, *including itself*. The following code will print `false`:

```
fmt.Println(math.NaN() == math.NaN())
```

To check that you got `NaN`, you need to use a special function such as `math.IsNaN`.

If you need better accuracy, look into `math/big` or external packages such as `shopspring/decimal`.

Further Reading

floating point zine by Julia Evans
http://twitter.com/b0rk/status/986424989648936960

What Every Computer Scientist Should Know About Floating-Point Arithmetic

http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Floating-Point Specification on Wikipedia
http://en.wikipedia.org/wiki/IEEE_754

Puzzle 8 Sleep Sort

sleep_sort.go

```
package main

import (
        "fmt"
        "sync"
        "time"
)

func main() {
        var wg sync.WaitGroup
        for _, n := range []int{3, 1, 2} {
                wg.Add(1)
                go func() {
                        defer wg.Done()
                        time.Sleep(time.Duration(n) * time.Millisecond)
                        fmt.Printf("%d ", n)
                }()
        }
        wg.Wait()
        fmt.Println()
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `2 2 2`

You probably expected `1 2 3`. Each goroutine sleeps `n` milliseconds and then prints `n`. `wg.Wait()` will wait until all goroutines are done, and then the program will print a newline.

However, the `n` that each goroutine uses is the same `n` defined in line 11. This is known as a *closure*.

This is the reason that, by design, goroutines (and deferred functions) are written as function invocation with parameters.

You have two options to solve this bug. The first, and my preference, is to pass `n` as a parameter to the goroutine.

sleep_sort_param.go

```
package main

import (
        "fmt"
        "sync"
        "time"
)

func main() {
        var wg sync.WaitGroup
        for _, n := range []int{3, 1, 2} {
                wg.Add(1)
                go func(i int) {
                        defer wg.Done()
                        time.Sleep(time.Duration(n) * time.Millisecond)
                        fmt.Printf("%d ", i)
                }(n)
        }
        wg.Wait()
        fmt.Println()
}
```

The second solution is to use the fact that every time you write `{` in Go, you open a new variable scope.

sleep_sort_scope.go

```
package main

import (
```

```
        "fmt"
        "sync"
        "time"
)

func main() {
        var wg sync.WaitGroup
        for _, n := range []int{3, 1, 2} {
                n := n // <1>
                wg.Add(1)
                go func() {
                        defer wg.Done()
                        time.Sleep(time.Duration(n) * time.Millisecond)
                        fmt.Printf("%d ", n)
                }()
        }
        wg.Wait()
        fmt.Println()
}
```

`n` is now a new variable that lives only in the `for` loop scope and, more importantly, in the goroutine closure. It "shadows" the outer `n` in line 11.

Further Reading

Function Closures in the Go Tour
http://tour.golang.org/moretypes/25

Closure on Wikipedia
http://en.wikipedia.org/wiki/Closure_(computer_programming

Puzzle 9 Just in Time

time_eq.go

```
package main

import (
        "encoding/json"
        "fmt"
        "log"
        "time"
)

func main() {
        t1 := time.Now()
        data, err := json.Marshal(t1)
        if err != nil {
                log.Fatal(err)
        }

        var t2 time.Time
        if err := json.Unmarshal(data, &t2); err != nil {
                log.Fatal(err)
        }
        fmt.Println(t1 == t2)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: **false**

You might expect this code to fail since there's no **time** type in the JSON format, or you might expect the comparison to succeed.

Go's `encoding/json` lets you define custom JSON serialization for types that are not supported by JSON. You do that by implementing `json.Marshaler` and `json.Unmarshaler` interfaces. Go's `time.Time` implements these interfaces by marshaling itself to an RFC 3339 formatted string and back.

JSON has a very limited set of types. For example, it only has floating-point numbers, which can lead to surprising results if you use the empty interface when unmarshaling.

json_float.go

```
package main

import (
        "encoding/json"
        "fmt"
        "log"
)

func main() {
        n1 := 1
        data, err := json.Marshal(n1)
        if err != nil {
                log.Fatal(err)
        }

        var n2 interface{}
        if err := json.Unmarshal(data, &n2); err != nil {
                log.Fatal(err)
        }

        fmt.Printf("n1 is %T, n2 is %T\n", n1, n2)
}
```

The previous will print `n1 is int, n2 is float64`.

Once you passed JSON serialization, you'd expect the times to be equal. The time package documentation says the following (my emphasis):

> Operating systems provide both a "wall clock," which is subject to changes for clock synchronization, and a "monotonic clock," which is not. The general rule is that the wall clock is for telling time and the monotonic clock is for measuring time. Rather than split the API, in this package *the Time returned by time.Now contains both a wall clock reading and a monotonic clock* reading …

Monotonic clocks are used for measuring duration. They exist to avoid problems such as your computer switching to daylight saving time during measurement. The value of a monotonic clock by itself does not mean anything; only the difference between two monotonic clock readings is useful.

When you use `==` to compare `time.Time`, Go will compare the `time.Time` struct fields, including the monotonic reading. However, when Go serializes a `time.Time` to JSON, it doesn't include the monotonic clock in the output. When you read back the time to `t2`, it doesn't contain the monotonic reading, and the comparison fails.

The solution to the problem is written in `time.Time`'s documentation:

> In general, prefer t.Equal(u) to t == u, since t.Equal uses the most accurate comparison available and correctly handles the case when only one of its arguments has a monotonic clock reading.

Further Reading

JSON Format
http://json.org

Falsehoods Programmers Believe About Time
http://infiniteundo.com/post/25326999628/falsehoods-programmers-believe-about-time

Time Package Documentation

http://golang.org/pkg/time/#Time

RFC 3339 in the below references and in the Index
http://ietf.org/rfc/rfc3339.txt

ISO 8601 on Wikipedia
http://en.wikipedia.org/wiki/ISO_8601

Puzzle 10 A Simple Append

append.go

```
package main

import (
        "fmt"
)

func main() {
        a := []int{1, 2, 3}
        b := append(a[:1], 10)
        fmt.Printf("a=%v, b=%v\n", a, b)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `a=[1 10 3], b=[1 10]`

You'll need to dig a bit into how slices are implemented and how `append` works to understand why you seed this output.

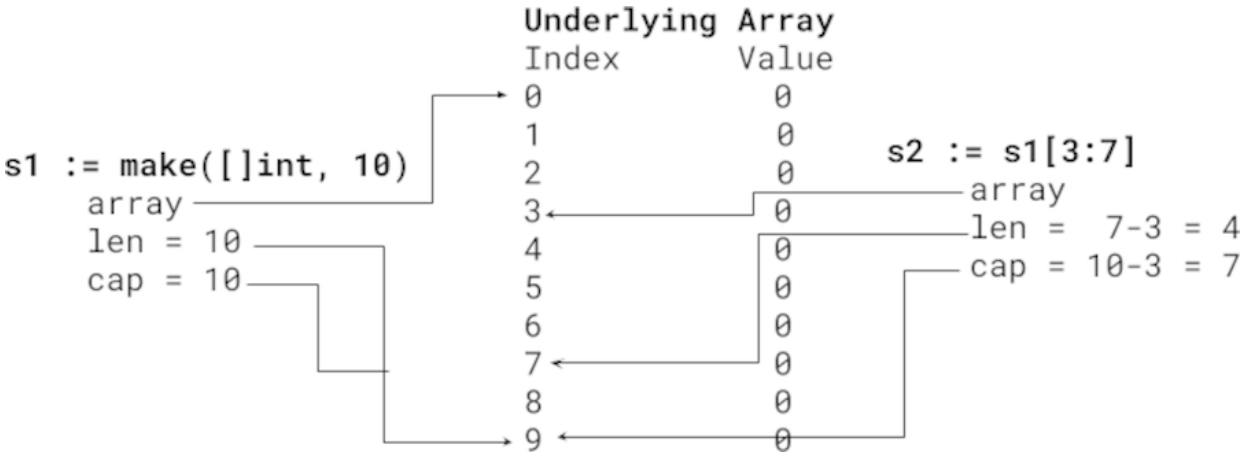
If you look at `src/runtime/slice.go` in the Go source code, you will see

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

`slice` is a struct with three fields. `array` is a pointer to the underlying array that holds the data. `len` is the length of the slice. `cap` is the capacity of the

underlying array.

Here's an illustrated example:



Slice notation (e.g., `s1[3:7]`) is *half open*, meaning from the first index up to but not including the last index. In `s2` you get indices 3, 4, 5, and 6, which means the length (`len`) is 4.

The capacity is how many items there are from the start of the slice to the end of the underlying array. In `s2` we start at 3 and can go up to 10, meaning the capacity is 7.

You can check length and capacity with the built-in `len` and `cap` functions. `len(s2)` is 4 and `cap(s2)` is 7.

The next piece you might be missing is how `append` works. When you call `append` it will check the capacity. If there is enough space, `append` will change the underlying array with the appended item and return a slice, pointing to the same array with bigger length. If there's no more space in the underlying array, `append` will create a new and bigger array, copy over the old array, update the new array with the new item, and return a slice pointing to the new array.

Here's a possible implementation of `append`:

```
    func Append(items []int, i int) []int {
```

```
        if len(items) == cap(items) { // No more space in underlying array
            // Go has a better growth heuristic than adding 1 every append
            newItems := make([]int, len(items)+1)
            copy(newItems, items)
            items = newItems
        } else {
            items = items[:len(items)+1]
        }

        items[len(items)-1] = i
        return items
    }
```

Now you can figure out output. The interesting line is `b := append(a[:1], 10)`. `a[:1]` creates a slice of length 1 and capacity of 3. `append` will find there's enough space and change the underlying array, placing `10` at index 1.

Both `a` and `b` point to the same underlying array. `a` has a length of 3, and `b` has a length of 2. That's why the output is `a=[1 10 3], b=[1 10]`.

Further Reading

Slices Internal in the Go Blog
http://blog.golang.org/slices

Slices in Effective Go
http://golang.org/doc/effective_go.html#slices

Slice Tricks in the Go Wiki
http://github.com/golang/go/wiki/SliceTricks

Puzzle 11 What's in a Log?

struct.go

```
package main

import (
        "fmt"
        "time"
)

// Log is a log message
type Log struct {
        Message string
        time.Time
}

func main() {
        ts := time.Date(2009, 11, 10, 0, 0, 0, 0, time.UTC)
        log := Log{"Hello", ts}
        fmt.Printf("%v\n", log)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `2009-11-10 00:00:00 +0000 UTC`

The `%v` verb will print all the struct fields.

struct_pt.go

```
package main
```

```
import (
        "fmt"
)

// Point is a 2D point
type Point struct {
        X int
        Y int
}

func main() {
        p := Point{1, 2}
        fmt.Printf("%v\n", p) // {1 2}
}
```

You'd expect the teaser code to print `{Hello 2009-11-10 00:00:00 +0000 UTC}`. The reason it doesn't is due to the way the `Log` struct is defined. In line 11 there is a field with no name, just a type. This is called *embedding*, and it means that the `Log` type has all the methods and fields that `time.Time` has.

`time.Time` defines a `String() string` method, which means it implements the [`fmt.Stringer` interface. And since `Log` embeds `time.Time` it also has a `String() string` method. If a parameter passed to `fmt` `fmt.Printf` implements `fmt.Stringer`, `fmt.Printf` will use it instead of the default output.

If you change the definition in line 11 to `Time time.Time`, you will see the expected output of `{Hello 2009-11-10 00:00:00 +0000 UTC}`.

You can also embed interfaces in Go. See the definition of `io.ReadWriter`, for example:

```
type ReadWriter interface {
    Reader
    Writer
}
```

Further Reading

Fun with Flags on the Gopher Academy Blog

http://blog.gopheracademy.com/advent-2019/flags/

Embedding in Effective Go
http://golang.org/doc/effective_go.html#embedding

Methods, Interfaces, and Embedded Types in Go on Ardan Labs Blog
http://ardanlabs.com/blog/2014/05/methods-interfaces-and-embedded-types.html

Puzzle 12 A Funky Number?

num.go

```
package main

import (
        "fmt"
)

func main() {
        fmt.Println(0x1p-2)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `0.25`

Go has several number types. The two main ones are

Integer
: These are whole numbers. Go has `int8`, `int16`, `int32`, `int64`, and `int`. There are also all the unsigned ones such as `uint8` and so on.

Float
: These are real numbers. Go has `float32` and `float64`.

There are other types such as `complex` and the various types defined in `math/big`.

When you write a number literal, such as `3.14`, the Go compiler needs to parse it to a specific type (`float64`, in this case). The Go spec[2] defines how you can write numbers. Let's have a look at some examples.

num_lit.go

```
package main

import (
        "fmt"
)

func main() {
        // Integer
        printNum(10)    // 10 of type int
        printNum(010)   // 8 of type int
        printNum(0x10)  // 16 of type int
        printNum(0b10)  // 2 of type int
        printNum(1_000) // 1000 of type int <1>

        // Float
        printNum(3.14)   // 3.14 of type float64
        printNum(.2)     // 0.2 of type float64
        printNum(1e3)    // 1000 of type float64
        printNum(0x1p-2) // 0.25 of type float64

        // Complex
        printNum(1i)     // (0+1i) of type complex128
        printNum(3 + 7i) // (3+7i) of type complex128
        printNum(1 + 0i) // (1+0i) of type complex128
}

func printNum(n interface{}) {
        fmt.Printf("%v of type %T\n", n, n)
}
```

`_` serves as the thousands separator. It makes big numbers much more readable for us humans.

`1e3` is known as *scientific notation*.

`0x1p-2` is called a *hexadecimal floating-point literal* in the Go specification and is following the IEEE 754 2008 specification. To calculate the value, do the following:

- Compute the value before the `p` as a hexadecimal number. In this example it's `0x1` = 1.
- Compute the value after the `p` as *2 to the power of that value*. In this example it's `2-2` = 0.25.
- Finally, multiply the two numbers, in this example, `1 * 0.25` = 0.25.

Further Reading

Lexical Elements in the Go Specification
http://golang.org/ref/spec#Lexical_elements

Scientific Notation on Wikipedia
http://en.wikipedia.org/wiki/Scientific_notation

IEEE 754 on Wikipedia
http://en.wikipedia.org/wiki/IEEE_754

Integer Literals
http://golang.org/ref/spec#Integer_literals

Floating-Point Literals
http://golang.org/ref/spec#Floating-point_literals

Imaginary Literals
http://golang.org/ref/spec#Imaginary_literals

Puzzle 13 Free-Range Integers

range.go

```
package main

import (
        "fmt"
)

func fibs(n int) chan int {
        ch := make(chan int)

        go func() {
                a, b := 1, 1
                for i := 0; i < n; i++ {
                        ch <- a
                        a, b = b, a+b
                }
        }()

        return ch
}

func main() {
        for i := range fibs(5) {
                fmt.Printf("%d ", i)
        }
        fmt.Println()
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will deadlock.

To get a value from a channel, you can either use the receive operator (`←ch`) or do a `for` loop with a `range` on the channel, consuming everything from it.

How does `range` know when there are no more values in the channel? It waits for the channel to be closed. The problem in the previous script is that the goroutine does not close the channel.

range_close.go

```
package main

import (
        "fmt"
)

func fibs(n int) chan int {
        ch := make(chan int)

        go func() {
                defer close(ch) // <1>
                a, b := 1, 1
                for i := 0; i < n; i++ {
                        ch <- a
                        a, b = b, a+b
                }
        }()

        return ch
}

func main() {
        for i := range fibs(5) {
                fmt.Printf("%d ", i)
        }
        fmt.Println()
}
```

What `range` does when iterating over a channel is something like this:

range_rcv.go

```
package main

import (
        "fmt"
)

func fibs(n int) chan int {
        ch := make(chan int)

        go func() {
                defer close(ch)
                a, b := 1, 1
                for i := 0; i < n; i++ {
                        ch <- a
                        a, b = b, a+b
                }
        }()

        return ch
}

func main() {
        ch := fibs(5)
        for {
                i, ok := <-ch
                if !ok {
                        break
                }
                fmt.Printf("%d ", i)
        }
        fmt.Println()
}
```

When you use this method to create iterators, you need to beware of goroutine leaks. If you create `ch := fibs(3)` and then consume only one or two values from it, the goroutine inside `fibs` will be blocked on sending to the channel. Since there's a reference to the channel, the garbage collector won't reclaim it. As Dave Cheney says, "Never start a goroutine without knowing how it will finish."

The common practice is to pass a `done` channel or a `context.Context`. This will complicate the code a bit, but you'll have control over stopping goroutines and will avoid goroutine leaks.

range_ctx.go

```
package main

import (
        "context"
        "fmt"
)

func fibs(ctx context.Context, n int) chan int {
        ch := make(chan int)
        go func() {
                defer close(ch)
                a, b := 1, 1
                for i := 0; i < n; i++ {
                        select {
                        case ch <- a:
                                a, b = b, a+b
                        case <-ctx.Done():
                                fmt.Println("cancelled")
                        }
                }
        }()

        return ch
}

func main() {
        ctx, cancel := context.WithCancel(context.Background())
        ch := fibs(ctx, 5)
        for i := 0; i < 3; i++ {
                val := <-ch
                fmt.Printf("%d ", val)
        }
        fmt.Println()
        cancel()
}
```

Further Reading

Go Concurrency Patterns: Context
http://blog.golang.org/context

Go Concurrency Patterns: Pipelines and Cancellation
http://blog.golang.org/pipelines

Package Context
http://golang.org/pkg/context/

"Concurrency" Chapter in Practical Go by Dave Cheney
http://dave.cheney.net/practical-go/presentations/gophercon-israel.html#concurrency

Puzzle 14 Multiple Personalities

multi_assign.go

```
package main

import (
        "fmt"
)

func main() {
        a, b := 1, 2
        b, c := 3, 4
        fmt.Println(a, b, c)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `1 3 4`

Go's short variable declaration (`:=`) can be used in a *multivalue* context, where there is more than one variable on the left side.

If all variables on the left are new, you're good:

```
a, b := 1, 2
```

If you have no new variables, you'll get a compilation error:

```
a, b := 1, 2
a, b := 2, 3 // error:  no new variables on left side of :=
```

When there's a mix, like in this teaser, you're still good as long as the types match. In this case, when you do

```
b, c := 3, 4
```

`b` is an existing variable, and the type of `3` that is `int` matches the current type of `b`. OK here. `c` is a new variable, making this statement OK.

Further Reading

Short Variable Declaration in the Go Tour
http://tour.golang.org/basics/10

Constant Declarations in the Go Specification
http://golang.org/ref/spec#Constant_declarations

Puzzle 15 A Tale of Two Cities

```
package main

import (
        "fmt"
)

func main() {
        city1, city2 := "Kraków", "Kraków"
        fmt.Println(city1 == city2)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

Don't try to copy and paste code from the PDF to your editor, it might or might not work. Use the files from the source repository to run the code.

This code will print: `false`

Your eyesight is OK. These two strings *look* the same. However, if you print the length of each variable, the length of `city1` will be 7 and the length of `city2` will be 8.

In the puzzle Puzzle 3, *When in Kraków* we talked about the fact that Go's strings are UTF-8 encoded byte slices. UTF-8 has many features, and some *characters* (runes) are not for display but for control, for example, text direction, sometimes called *bidi* (short for bidirectional).

`city1` has a 1-byte rune at position 4 (`ó`), while `city2` has the `o` rune in position 4 and a control character saying "add an umlaut to the previous character."

These ways of encoding are called NFC and NFD.

When you compare strings, they are compared at the byte level. This is why you'll see `false` as the output. To fix this you need to use an external library called `golang.org/x/text/unicode/norm`.

two_cities_nfc.go

```
package main

import (
        "fmt"

        "golang.org/x/text/unicode/norm"
)

func main() {
        city1, city2 := "Kraków", "Kraków"
        city1, city2 = norm.NFC.String(city1), norm.NFC.String(city2)
        fmt.Println(city1 == city2)
}
```

Make sure to normalize all the text you're working with to a single form (e.g., NFC).

The `golang.org/x` packages are not in the standard library but are maintained by the Go core developers (and friends). There are many useful libraries under `golang.org/x`. I recommend investing some time in getting to know them.

For example, `golang.org/x/text/transform` has a `NewReader` function that returns an `io.Reader` that will convert everything it reads to a normal form.

Further Reading

Bidi on Wikipedia
http://en.wikipedia.org/wiki/Bidirectional_text

Unicode Equivalence on Wikipedia
http://en.wikipedia.org/wiki/Unicode_equivalence

Strings, Bytes, Runes, and Characters in Go
http://blog.golang.org/strings

UTF-8 on Wikipedia
http://en.wikipedia.org/wiki/UTF-8

Puzzle 16 What's in a Channel?

chan.go

```
package main

import (
        "fmt"
)

func main() {
        ch := make(chan int, 2)
        ch <- 1
        ch <- 2
        <-ch
        close(ch)
        a := <-ch
        b := <-ch
        fmt.Println(a, b)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `2 0`

`ch` is a buffered channel with a capacity of 2 (you can check this with `cap(ch)`). You send two values to `ch`, 1 and 2. If you try to send another one, you'll get a deadlock.

You then receive a value from `ch`, close it, and receive two more values from it. The question is, what happens when you receive from a closed channel?

The answer is that if there are values in the buffer, you will get them; otherwise, you will get the zero value for the channel type. This is why `a` gets the value of 2, which was in the buffer, and `b` gets 0, which is the zero value for `int`.

How can you know if a value you get from a channel is a value that was actually there or a zero value since the channel was closed? With the usual `comma, ok` paradigm.

chan_empty.go

```
package main

import (
        "fmt"
)

func main() {
        ch := make(chan int, 1)
        ch <- 1
        a, ok := <-ch
        fmt.Println(a, ok) // 1 true
        close(ch)
        b, ok := <-ch
        fmt.Println(b, ok) // 0 false
}
```

Further Reading

Buffered Channels in the Go Tour
http://tour.golang.org/concurrency/3

Channels in Effective Go
http://golang.org/doc/effective_go.html#channels

Why Are There nil Channels in Go?
http://medium.com/justforfunc/why-are-there-nil-channels-in-go-9877cc0b2308

Puzzle 17 An **Int**eresting String

strint.go

```
package main

import (
        "fmt"
)

func main() {
        i := 169
        s := string(i)
        fmt.Println(s)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: ©

The `string` type supports type conversion from `int`. It'll treat this integer as a rune. The rune `169` is the copyright sign (©). People who make this mistake usually come from languages such as Python where `str(169)` → `"169"`.

If you want to convert a number to a string (or a string to a number), use the `strconv` package.

strint_conv.go

```
package main

import (
```

```
        "fmt"
        "strconv"
)

func main() {
        i := 169
        s := strconv.Itoa(i)
        fmt.Println(s)
}
```

Strings also support type conversion from a byte slice.

strint_byte.go

```
package main

import (
        "fmt"
)

func main() {
        data := []byte{'1', '2', '9'}
        s := string(data)
        fmt.Println(s) // 129
}
```

When you convert from a byte slice to a string, Go will copy the byte slice, which does a memory allocation. In maps, where you can't use a `[]byte` as a key, there is a compiler optimization.

strint_map.go

```
package main

import (
        "fmt"
)

func main() {
        m := map[string]int{
                "hello": 3,
        }
```

```
        key := []byte{'h', 'e', 'l', 'l', 'o'}
        val := m[string(key)] // no memory allocation
        fmt.Println(val)      // 3
}
```

Further Reading

strconv *Documentation*

http://golang.org/pkg/strconv/

runenames in ***golang.org/x***

http://godoc.org/golang.org/x/text/unicode/runenames

"Using []byte as a Map Key" from Dave Cheney's "High-Performance Go Workshop"

https://dave.cheney.net/high-performance-go-workshop/gophercon-2019.html

Puzzle 18 A Job to Do

job.go

```
package main

import (
        "fmt"
)

type Job struct {
        State string
        done  chan struct{}
}

func (j *Job) Wait() {
        <-j.done
}

func (j *Job) Done() {
        j.State = "done"
        close(j.done)
}

func main() {
        ch := make(chan Job)
        go func() {
                j := <-ch
                j.Done()
        }()

        job := Job{"ready", make(chan struct{})}
        ch <- job
        job.Wait()
        fmt.Println(job.State)
}
```

Guess the Output

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `ready`

At first glance, it looks like the code is OK. You're using a pointer receiver in the `Job` struct methods. The fact that the call to `Wait` terminated tells you that the channel was closed.

The problem is with the definition of `ch`. It is a channel of `Job`, not `*Job`, which means that when you send the variable `job` over the channel, you actually send a copy of it. A channel in Go is a *pointer-like* type, so even though there is a copy of `job` inside the goroutine, `j.done` points to the same channel `job.done` is pointing to.

Strings in Go are not pointer-like. When you call `j.Done()`, the string inside the goroutine, you change the value of the `State` field in the goroutine copy of `job`. This change is not reflected in the `job` variable declared in line 28.

The solution is to make `ch` type `*Job`.

job_ptr.go

```
package main

import (
        "fmt"
)

type Job struct {
        State string
        done  chan struct{}
}
```

```
func (j *Job) Wait() {
        <-j.done
}

func (j *Job) Done() {
        j.State = "done"
        close(j.done)
}

func main() {
        ch := make(chan *Job)
        go func() {
                j := <-ch
                j.Done()
        }()

        job := Job{"ready", make(chan struct{})}
        ch <- &job
        job.Wait()
        fmt.Println(job.State)
}
```

Further Reading

There Is No Pass-by-Reference in Go
http://dave.cheney.net/2017/04/29/there-is-no-pass-by-reference-in-go

Channel Types Specification
http://golang.org/ref/spec#Channel_types

Go Concurrency Patterns: Pipelines and Cancellation
http://blog.golang.org/pipelines

Channels in the Go Tour
http://tour.golang.org/concurrency/2

Puzzle 19 To Err or Not to Err

error.go

```
package main

import (
        "fmt"
)

type OSError int

func (e *OSError) Error() string {
        return fmt.Sprintf("error #%d", *e)
}

func FileExists(path string) (bool, error) {
        var err *OSError
        return false, err // TODO
}

func main() {
        if _, err := FileExists("/no/such/file"); err != nil {
                fmt.Printf("error: %s\n", err)
        } else {
                fmt.Println("OK")
        }
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: **error: <nil>**

The `if` statement in line 19 says that `err != nil`, but `err` prints out as `<nil>`. Furthermore, `err` in line 14 is not initialized, meaning it has the zero value for a pointer, which *is* `nil`.

If you look at `src/runtime/runtime2.go` in the Go source repository, you'll see the following definition:

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```

- `itab` describes the interface.
- `data` is a pointer to the value that implements the interface.

An interface is considered `nil` only if both `itab` and `data` are `nil`, which is not the case here.

The solution is *always* to use variables of type `error` when returning errors. In this case, change line 14 to `var err error`.

Further Reading

Interfaces in go-internals Book
http://github.com/teh-cmc/go-internals/blob/master/chapter2_interfaces/README.md

Go Data Structures: Interfaces
http://research.swtch.com/interfaces

Puzzle 20 What's in a String?

runes.go

```
package main

import (
        "fmt"
)

func main() {
        msg := "π = 3.14159265358..."
        fmt.Printf("%T ", msg[0])
        for _, c := range msg {
                fmt.Printf("%T\n", c)
                break
        }
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `uint8 int32`

Go strings are UTF-8 encoded. When you access a string with `[]` or with `len`, Go will access the underlying `[]byte`. Slice is a type in Go that's aliased to `uint8`.

When you iterate over a string in Go, you will get the Unicode character called *rune*, which is an alias to `int32`. This is because the characters in UTF-8 can be up to 4 bytes.

If you print a rune using the `%v` verb, you'll see the numeric value. Use the `%c` verb to display the character. If you see `?` or other characters instead of the rune you're trying to print, it means the current font does not support it. There's no single font that can display all fonts in the Unicode specification.

There are several ways to write a rune literal. Let's have a look in the following section.

runes_lit.go

```
package main

import (
        "fmt"
)

func main() {
        r1 := '™'
        fmt.Println(r1)        // 8482
        fmt.Printf("%c\n", r1) // ™

        r2 := '\x61'           // \x - 2 digits
        fmt.Printf("%c\n", r2) // a

        r3 := '\u2122'         // \u - 4 digits (8482 in hex)
        fmt.Printf("%c\n", r3) // ™

        r4 := '\U00002122'     // \U - 8 digits
        fmt.Printf("%c\n", r4) // ™
}
```

Further Reading

Strings, Bytes, Runes, and Characters in Go
http://blog.golang.org/strings

Unicode and You
http://betterexplained.com/articles/unicode/

Unicode on Wikipedia
http://en.wikipedia.org/wiki/Unicode

UTF-8 on Wikipedia
http://en.wikipedia.org/wiki/UTF-8

Rune Literals in the Go Specification
http://golang.org/ref/spec#Rune_literals

Unicode Character Table
http://unicode-table.com/

Puzzle 21 A Double Take

init.go

```
package main

import (
        "fmt"
)

func init() {
        fmt.Printf("A ")
}

func init() {
        fmt.Print("B ")
}

func main() {
        fmt.Println()
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: **A B**

Normally, the Go compiler will not let you define two functions with the same name in the same package. However, `init` is special. Here's what the documentation says:

> Multiple such functions may be defined per package, even within a single source file. In the package block, the `init` identifier can be used

> only to declare `init` functions, yet the identifier itself is not declared. Thus, `init` functions cannot be referred to from anywhere in a program.

`init` is the final stage in package initialization after imported modules are initialized and package-level variables are initialized.

Since you can initialize package-level variables with function calls, save `init` for special cases. Also, try to avoid package-level variables as much as you can. I seldom write `init` functions and prefer to do initialization in `main` where I have better control over order of initialization, error handling, and logging.

One of the problems with package-level variables or `init` is that you can't return error values. If you have an error in `init`, the best course of action is to panic and not continue execution in a bad state. To support that, some of the Go packages provide a `Must` version of their `New` functions. For example, the `regexp` package has `MustCompile`:

> MustCompile is like Compile but panics if the expression cannot be parsed. It simplifies safe initialization of global variables holding compiled regular expressions.

Try to think about your types and whether you should provide a `Must` function for them as well.

Further Reading

init Function in Effective Go
http://golang.org/doc/effective_go.html#init

Package Initialization in the Go Specification
http://golang.org/ref/spec#Package_initialization

"Removing Package Scoped Variables, in Practice" by Dave Cheney

http://dave.cheney.net/2017/06/11/go-without-package-scoped-variables

Writing Deployable Code (Part 2) by Ran Tavory
http://medium.com/@rantav/writing-deployable-code-part-two-217bc884c917

Puzzle 22 Count Me a Million

count.go

```
package main

import (
        "fmt"
        "sync"
)

func main() {
        var count int
        var wg sync.WaitGroup

        for i := 0; i < 1_000_000; i++ {
                wg.Add(1)
                go func() {
                        defer wg.Done()
                        count++
                }()

        }
        wg.Wait()
        fmt.Println(count)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `891239`

Different Output

Different Output

Your output might be different, but it will be below 1,000,000.

What you have here is a race condition. The operation to increment an integer is not atomic, meaning the code can get interrupted mid-operation. On top of that, there is the memory barrier. To speed things up, modern computers try to fetch values from a CPU cache before going the long way to main memory.

You can look at the assembly output of `count.go` by running `go tool compile -S count.go`. When you do that, you will see the code generated for `count++` on line 16:

```
0x0045 00069 (count.go:16)  PCDATA  $0, $1
0x0045 00069 (count.go:16)  PCDATA  $1, $4
0x0045 00069 (count.go:16)  MOVQ    "".&count+56(SP), AX
0x004a 00074 (count.go:16)  PCDATA  $0, $0
0x004a 00074 (count.go:16)  INCQ    (AX)
```

`MOVQ` will fetch the current value of the counter to the `AX` register. This might come from the memory or from the cache.

The Go toolchain can help you detect such race conditions. Both `go run` and `go test` have a `-race` flag to detect race conditions. You can try it out:

```
$ go run -race count.go
==================
WARNING: DATA RACE
Read at 0x00c00011a010 by goroutine 8:
        main.main.func1()
                        ./code/count.go:16 +0x6c

Previous write at 0x00c00011a010 by goroutine 7:
        main.main.func1()
                        ./code/count.go:16 +0x82
```

```
Goroutine 8 (running) created at:
        main.main()
                        ./code/count.go:14 +0xe4

Goroutine 7 (finished) created at:
        main.main()
                        ./code/count.go:14 +0xe4
```

To solve this problem, you can use a sync.Mutex to ensure only one goroutine updates count at a time.

count_mu.go

```
package main

import (
        "fmt"
        "sync"
)

func main() {
        var count int
        var wg sync.WaitGroup
        var m sync.Mutex

        for i := 0; i < 1_000_000; i++ {
                wg.Add(1)
                go func() {
                        m.Lock()
                        defer m.Unlock()
                        defer wg.Done()
                        count++
                }()

        }
        wg.Wait()
        fmt.Println(count)
}
```

Another option is to use the sync/atomic package. sync/atomic, provides atomic operations that are faster than using a mutex but harder to use.

count_atomic.go

```
package main

import (
        "fmt"
        "sync"
        "sync/atomic"
)

func main() {
        var count int64 // Atomic works with int64, not int
        var wg sync.WaitGroup

        for i := 0; i < 1_000_000; i++ {
                wg.Add(1)
                go func() {
                        defer wg.Done()
                        atomic.AddInt64(&count, 1)
                }()

        }
        wg.Wait()
        fmt.Println(count)
}
```

Even though Go tries to abstract the hardware away, you still need to learn it and understand it.

Further Reading

Race Condition on Wikipedia
http://en.wikipedia.org/wiki/Race_condition

Atomic vs. Non-atomic Operations
http://preshing.com/20130618/atomic-vs-non-atomic-operations/

Introducing the Go Race Detector
http://blog.golang.org/race-detector

Memory Barriers / Fences

http://mechanical-sympathy.blogspot.com/2011/07/memory-barriersfences.html

Memory Barrier on Wikipedia
http://en.wikipedia.org/wiki/Memory_barrier

Scheduling in Go: Part I – OS Scheduler
http://ardanlabs.com/blog/2018/08/scheduling-in-go-part1.html

Computer Latency at a Human Scale
http://twitter.com/srigi/status/917998817051541504

Puzzle 23 Who's Next?

next_id.go

```
package main

import (
        "fmt"
)

var (
        id = nextID()
)

func nextID() int {
        id++
        return id
}

func main() {
        fmt.Println(id)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will not compile.

Building the code shows the problem:

```
$ go build next_id.go
# command-line-arguments
./next_id.go:8:2: initialization loop:
        /home/miki/Projects/go-brain-teasers/code/next_id.go:8:2: id
refers to
```

```
/home/miki/Projects/go-brain-teasers/code/
next_id.go:11:6: nextID refers to
/home/miki/Projects/go-brain-teasers/code/next_id.go:8:2: id
```

There is an initialization loop in the code. The value of `id` depends on `nextID`, which uses the value of `id`, which …

The Go compiler tries to find these loops, but there are cases in which it cannot detect them (called *hidden dependencies*).

Make sure your package initialization is simple and easy to understand. Try to avoid package-level variables, and defer initialization to `main` as much as possible.

Further Reading

Package Initialization in the Go Specification
http://golang.org/ref/spec#Package_initialization

"Removing Package Scoped Variables, in Practice" by Dave Cheney
http://dave.cheney.net/2017/06/11/go-without-package-scoped-variables

Puzzle 24 Fun with Flags

flag.go

```
package main

import (
        "fmt"
)

type Flag int

func main() {
        var i interface{} = 3
        f := i.(Flag)
        fmt.Println(f)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will panic.

Even though `Flag` is basically an `int`, the Go compiler sees it as a distinct type, and the type assertion (`i.(Flag)`) will panic. This is another case where you can use the **comma, ok** paradigm.

flag_ok.go

```
package main

import (
        "fmt"
)
```

```
type Flag int

func main() {
        var i interface{} = 3
        f, ok := i.(Flag)
        if !ok {
                fmt.Println("not a Flag")
                return
        }
        fmt.Println(f)
}
```

You can also change line 7 to `type Flag = int`. Now `Flag` is a type alias and the code will work. However, now you can use an `int` whenever you need a `Flag` and the type system won't protect you.

Further Reading

Type Aliases
http://golang.org/doc/go1.9#language

"Codebase Refactoring (with Help from Go)" (explains the rationale for type aliases)
http://talks.golang.org/2016/refactor.article

Type Declarations in the Go Specification
http://golang.org/ref/spec#Type_declarations

Type Assertions in the Go Specification
http://golang.org/ref/spec#Type_assertions

Type Switch in Effective Go
http://golang.org/doc/effective_go.html#type_switch

Interface Conversions and Type Assertions in Effective Go
http://golang.org/doc/effective_go.html#interface_conversions

Puzzle 25 You Have My Permission

iota.go

```
package main

import (
        "fmt"
)

const (
        Read = 1 << iota
        Write
        Execute
)

func main() {
        fmt.Println(Execute)
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

This code will print: `4`

`iota` is Go's version of an enumerated type. It can be used inside a `const` declaration. `iota` grows by one for each constant in the same group. If you specify an operation on `iota` (e.g., addition), it'll carry on. `<<` is the left shift operator. It'll move the bits in a number `n` places to the left:

- `1` in binary (8 bit) is `00000001`.
- `1<<3` is `00001000`, which is 8.

In this case, we start with `Read = 1<<iota`, which translates to `1<<0 → 0001`, which is 1. Then there's an implicit `1<<1 → 0010` for `Write`, which is 2, and finally, an implicit `1<<2 → 0100` for `Execute`, which is 4.

You'll mostly use these bit operations for flags. You can pack eight flags in one byte using `|` (bitwise OR) (called *bitmask*) and efficiently check whether a flag is set with `&` (bitwise AND).

iota_check.go

```
package main

import (
        "fmt"
)

const (
        Read = 1 << iota
        Write
        Execute
)

func main() {
        mask := Read | Execute

        if mask&Execute == 0 {
                fmt.Println("can't execute")
        } else {
                fmt.Println("can execute") // will be printed
        }

        if mask&Write == 0 {
                fmt.Println("can't write") // will be printed
        } else {
                fmt.Println("can write")
        }
}
```

`iota` values are numbers. In some cases you'd like your constants to have a human-readable representation. This is done by giving these constants a

type and implementing the `fmt.Stringer` interface for this type.

iota_str.go

```
package main

import (
        "fmt"
)

type FilePerm uint16 // 16 flags are enough

const (
        Read FilePerm = 1 << iota
        Write
        Execute
)

// String implements fmt.Stringer interface
func (p FilePerm) String() string {
        switch p {
        case Read:
                return "read"
        case Write:
                return "write"
        case Execute:
                return "execute"
        }

        return fmt.Sprintf("unknown FilePerm: %d", p) // don't use %s
here :)
}

func main() {
        fmt.Println(Execute)        // execute
        fmt.Printf("%d\n", Execute) // 4
}
```

I'll leave it as an exercise for you to implement a `String() string` method that supports bitmasks. ☺

Further Reading

Enumerated Type on Wikipedia

http://en.wikipedia.org/wiki/Enumerated_type

iota

http://github.com/golang/go/wiki/Iota

Constants in Effective Go

http://golang.org/doc/effective_go.html#constants

Bitwise Operation on Wikipedia

http://en.wikipedia.org/wiki/Bitwise_operation

Footnotes

[1] https://golang.org/ref/spec#String_literals

[2] https://golang.org/ref/spec#Lexical_elements

Thank you!
How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line “Book Feedback.”

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2021 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn’t apply to *The Pragmatic Programmer* ebook because it’s older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

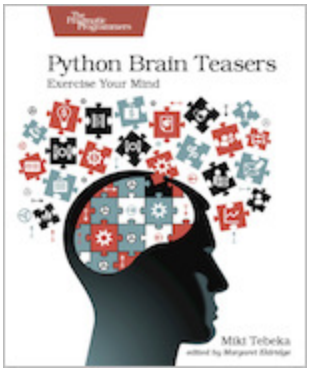
And thank you for your continued support,

The Pragmatic Bookshelf

You May Be Interested In…

Select a cover for more information

Python Brain Teasers

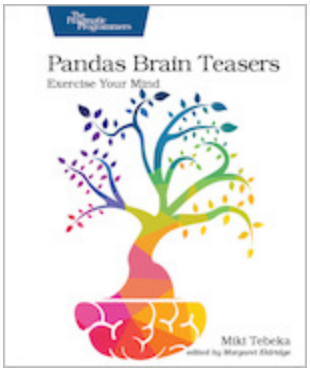


We geeks love puzzles and solving them. The Python programming language is a simple one, but like all other languages it has quirks. This book uses those quirks as teaching opportunities via 30 simple Python programs that challenge your understanding of Python. The teasers will help you avoid mistakes, see gaps in your knowledge, and become better at what you do. Use these teasers to impress your co-workers or just to pass the time in those boring meetings. Teasers are fun!

Miki Tebeka

(116 pages) ISBN: 9781680509007 $18.95

Pandas Brain Teasers



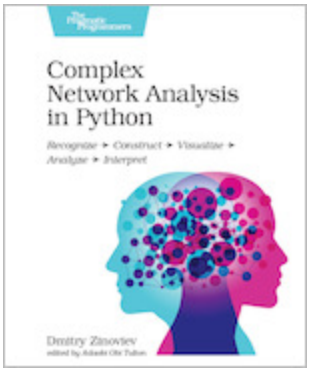
This book contains 25 short programs that will challenge your understanding of Pandas. Like any big project, the Pandas developers had to make some design decisions that at times seem surprising. This book uses those quirks as a teaching opportunity. By understanding the gaps in your knowledge, you'll become better at what you do. Some of the teasers are from the author's

experience shipping bugs to production, and some from others doing the same. Teasers and puzzles are fun, and learning how to solve them can teach you to avoid programming mistakes and maybe even impress your colleagues and future employers.

Miki Tebeka

(110 pages) ISBN: 9781680509014 $18.95

Complex Network Analysis in Python



Construct, analyze, and visualize networks with networkx, a Python language module. Network analysis is a powerful tool you can apply to a multitude of datasets and situations. Discover how to work with all kinds of networks, including social, product, temporal, spatial, and semantic networks. Convert almost any real-world data into a complex network—such as recommendations on co-using cosmetic products, muddy hedge fund connections, and online friendships. Analyze and visualize the network, and make business decisions based on your analysis. If you're a curious Python programmer, a data scientist, or a CNA specialist interested in mechanizing mundane tasks, you'll increase your productivity exponentially.

Dmitry Zinoviev

(260 pages) ISBN: 9781680502695 $35.95

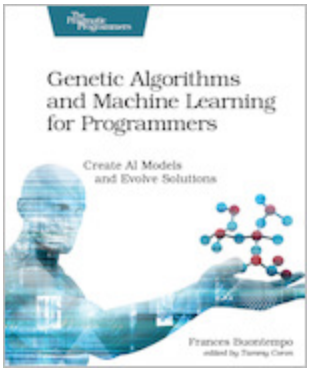
Intuitive Python

Developers power their projects with Python because it emphasizes readability, ease of use, and access to a meticulously maintained set of packages and tools. The language itself continues to improve with every release: writing in Python is full of possibility. But to maintain a successful Python project, you need to know more than just the language. You need tooling and instincts to help you make the most out of what's available to you. Use this book as your guide to help you hone your skills and sculpt a Python project that can stand the test of time.

David Muller

(140 pages) ISBN: 9781680508239 $26.95

Genetic Algorithms and Machine Learning for Programmers

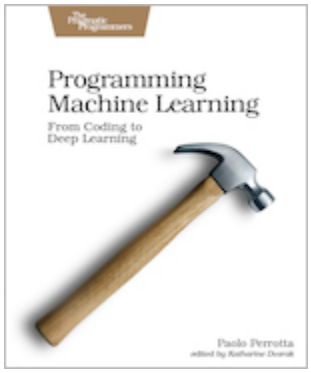


Self-driving cars, natural language recognition, and online recommendation engines are all possible thanks to Machine Learning. Now you can create your own genetic algorithms, nature-inspired swarms, Monte Carlo simulations, cellular automata, and clusters. Learn how to test your ML code and dive into even more advanced topics. If you are a beginner-to-intermediate programmer keen to understand machine learning, this book is for you.

Frances Buontempo

(234 pages) ISBN: 9781680506204 $45.95

Programming Machine Learning



You've decided to tackle machine learning — because you're job hunting, embarking on a new project, or just think self-driving cars are cool. But where to start? It's easy to be intimidated, even as a software developer. The good news is that it doesn't have to be that hard. Master machine learning by writing code one line at a time, from simple learning programs all the way to a true deep learning system. Tackle the hard topics by breaking them down so they're easier to understand, and build your confidence by getting your hands dirty.

Paolo Perrotta

(340 pages) ISBN: 9781680506600 $47.95

Hands-on Rust

Rust is an exciting new programming language combining the power of C with memory safety, fearless concurrency, and productivity boosters—and what better way to learn than by making games. Each chapter in this book presents hands-on, practical projects ranging from "Hello, World" to building a full dungeon crawler game. With this book, you'll learn game development skills applicable to other engines, including Unity and Unreal.

Herbert Wolverson

(342 pages) ISBN: 9781680508161 $47.95

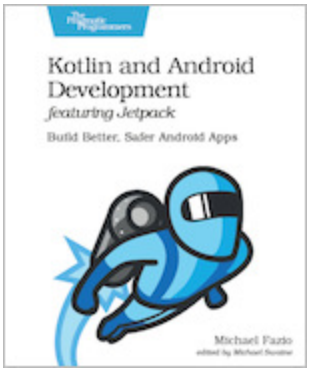
Python Testing with pytest, Second Edition

Test applications, packages, and libraries large and small with pytest, Python's most powerful testing framework. pytest helps you write tests quickly and keep them readable and maintainable. In this fully revised edition, explore pytest's superpowers—simple asserts, fixtures, parametrization, markers, and plugins—while creating simple tests and test suites against a small database application. Using a robust yet simple fixture model, it's just as easy to write small tests with pytest as it is to scale up to complex functional testing. This book shows you how.

Brian Okken

(250 pages) ISBN: 9781680508604 $45.95

Kotlin and Android Development featuring Jetpack



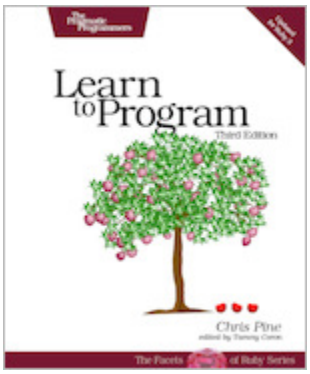
Start building native Android apps the modern way in Kotlin with Jetpack's expansive set of tools, libraries, and best practices. Learn how to create efficient, resilient views with Fragments and share data between the views with ViewModels. Use Room to persist valuable data quickly, and avoid NullPointerExceptions and Java's verbose expressions with Kotlin. You can even handle asynchronous web service calls elegantly with Kotlin coroutines.

Achieve all of this and much more while building two full-featured apps, following detailed, step-by-step instructions.

Michael Fazio

(444 pages) ISBN: 9781680508154 $49.95

Learn to Program, Third Edition

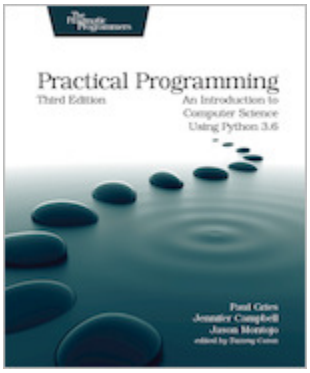


It's easier to learn how to program a computer than it has ever been before. Now everyone can learn to write programs for themselves—no previous experience is necessary. Chris Pine takes a thorough, but lighthearted approach that teaches you the fundamentals of computer programming, with a minimum of fuss or bother. Whether you are interested in a new hobby or a new career, this book is your doorway into the world of programming.

Chris Pine

(230 pages) ISBN: 9781680508178 $45.95

Practical Programming, Third Edition



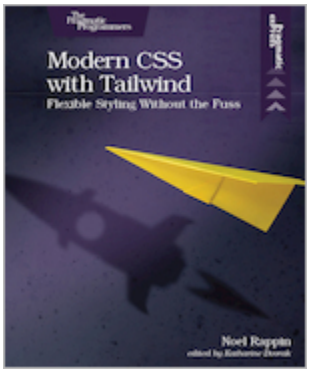
Classroom-tested by tens of thousands of students, this new edition of the best-selling intro to programming book is for anyone who wants to understand computer science. Learn about design, algorithms, testing, and debugging. Discover the fundamentals of programming with Python 3.6—a language that's used in millions of devices. Write

programs to solve real-world problems, and come away with everything you need to produce quality code. This edition has been updated to use the new language features in Python 3.6.

Paul Gries, Jennifer Campbell, Jason Montojo

(410 pages) ISBN: 9781680502688 $49.95

Modern CSS with Tailwind



Tailwind CSS is an exciting new CSS framework that allows you to design your site by composing simple utility classes to create complex effects. With Tailwind, you can style your text, move your items on the page, design complex page layouts, and adapt your design for devices from a phone to a wide-screen monitor. With this book, you'll learn how to use the Tailwind for its flexibility and its consistency, from the smallest detail of your typography to the entire design of your site.

Noel Rappin

(90 pages) ISBN: 9781680508185 $26.95